Operating System Protection Through Program Evolution

by Dr. Frederick B. Cohen ‡

In this paper, we introduce the use of program evolution as a technique for defending against automated attacks on operating systems.

Search terms: Trusted Systems, Computational Complexity, Program Evolution, Operating Systems

‡ This research was funded by ASP, PO Box 81270, Pittsburgh, PA 15217, USA

1 Background and Introduction

From the beginning of electronic computing until 15 years ago, the 'game' of attack and defense was played on a system by system basis, with defenders relying on physical security and ad-hoc operating system protection methods and attackers guessing passwords or exploiting errors and omissions to bypass normal system controls. Over the last 15 years, the computing environment has changed dramatically, with widespread physical distribution of computing power, almost complete loss of physical control over computing hardware, and a dramatic increase in the networking of computers, but defenses have not changed substantially in terms of their reliance on physical security and ad-hoc defenses. As a result, we see new classes of attacks such as computer viruses, which exploit the lack of physical control and fundamental weakness of existing logical controls to spread transitively throughout the computing world. Even the best protection systems available today can quickly and easily be defeated by anyone with physical access and ample knowledge, or by the application of that expertise in a widespread computer virus attack.

One of the major factors in the successful application of information protection techniques is the exploitation of computational advantage. Computational advantage shows up historically in cryptography, where Shannon's theory [?] clearly demonstrates the effect of 'workload' on the complexity of cryptanalysis and introduces the concepts of diffusion and confusion as they relate to statistical attacks on cryptosystems. Most modern cryptosystems exploit this as their primary defense [?] [?]. The same basic principle applies in computer virus analysis [?] [?] in which evolutionary viruses drive the complexity of detection and eradication up dramatically and in password protection in which we try to drive the number of guesses required for a successful attack up by limiting the use of obvious passwords [?]. As we will see, one of the major reasons attacks succeed is because of the static nature of defense, and the dynamic nature of attack.

1.1 The Ultimate Attack

The ultimate attack against any system begins with physical access, and proceeds to disassembly and reverse engineering of whatever programmed defenses are in place. Even with a cryptographic key provided by the user, an attacker can modify the mechanism to examine and exploit the key, given ample physical access. Eventually, the attacker can remove the defenses by finding decision points and altering them to yield altered decisions.

Without physical protection, nobody has ever found a defense against this attack, and it is unlikely that anyone ever will. The reason is that any protection scheme other than a physical one depends on the operation of a finite state machine, and ultimately, any finite state machine can be examined and modified at will, given enough time and effort. The best we can ever do is delay attack by increasing the complexity of making desired alterations.

With any static defense in widespread use, attackers can and may eventually find a technique to bypass protection and incorporate the technique in a virus. In the modern environment, this means that a virus writer can eventually exploit such a weakness to evade protection on a large number of systems. This has already taken place to some degree, with attackers producing computer viruses designed to exploit low-level operating system assumptions and bypass specific defensive products and techniques. [**?**]

To the extent that defenses have generic weaknesses, this problem will continue unabated, but even in the cases of defenses with no known generic weaknesses, attackers may succeed by using the ultimate attack and programming the results into a virus. Since the program that runs first 'wins' [1], and a virus in the boot block of a floppy disk runs first on most modern computer systems, an attacker might exploit knowledge gained from an ultimate attack on one system by placing the successful technique in a boot block initiated virus, and carry the attack to numerous machines. In the case of low-level viruses operating on IBM compatible PCs, the same attacks that work against the DOS operating system (e.g. The 'Stoned' virus and its variants) succeed even when the Unix operating system is used.

1.2 The Ultimate Defense

The ultimate defense is to drive the complexity of the ultimate attack up so high that the cost of attack is too high to be worth performing. This is, in effect, security through obscurity, and it is our general conclusion that all technical information protection in computer systems relies at some level either on physical protection, security through obscurity, or combinations thereof.

The goal of security through obscurity is to make the difficulty of attack so great that in practice, it is not worth performing, even though it could eventually be successful. Successful attacks against obscurity defenses depend on the ability to guess some key piece of information. The most obvious example is attacking and defending passwords, and since this problem demonstrates precisely the issues at hand, we will use it as an example. In password protection, there are generally three aspects to making attack difficult. One aspect is making the size of the password space large, so that the potential number of guesses required for an attack is enormous. The second aspect is spreading the probability density out so that there is relatively little advantage to searching the space selectively. This is basically the same as Shannon's concept of diffusion. The third aspect is obscuring the stored password

[1] i.e. it can simulate the hardware and alter or examine any information developed during operation, forge or prevent auditing, and deny services at will

information so that the attacker cannot simply read it in stored form. This is basically the same as Shannon's concept of confusion.

In successful password attacks, the size of the password space is usually substantial, but the way people select passwords leads to very small high probability subspaces. For example, the user's identification to the system spelled backwards is a very common authentication string. In numerous studies performed over many years, the vast majority of user's passwords could be guessed in only a few minutes [**?**]. The third aspect of password protection usually involves using access controls in combination with trapdoor encryption to drive up the time required to try substantial numbers of guesses.

In the case of current operating systems, the size of the space is enormous, consisting of all programs that fit in the computer's memory, but the operating system may only have a very small number of versions, all of which are almost identical, yielding a highly coherent subspace consisting of only a few very closely tied points. Thus, only a few guesses are required to determine precisely which version of the operating system is in use, and even that may not be required for many attacks. Operating systems also provide no confusion in that the part of the program that performs any given operation is immediately apparent to the knowledgeable attacker. For this reason, low-level operating system attack is quite simple.

The problem for defenders is to find a way to increase the difficulty of operating system attack by reducing coherence. The ultimate goal is to obscure defenses so as to make attackers require repeated use of the ultimate attack in order to impact substantial numbers of systems. One solution is to provide each computer with a sound and unique defense. This would tend to make it infeasible to design an automated attack that could systematically bypass all of the defenses, but then we would have to design a new defense for each system, and the costs of defense would probably become intolerable. We could trade off costs for probability of attack by implementing some fixed number of different defenses, thus requiring a fixed number of ultimate attacks for complete success. This is essentially the situation in the world today, where a wide variety of ad-hoc defenses are on the market. Unfortunately, many of these defenses fall to the same classes of attack, and the number is sufficiently small that defeating most of them requires relatively little effort or expense.

A more practical solution to this problem might be the use of evolutionary defenses. In order to make such a defensive strategy cost effective for numerous variations (e.g. one per computer worldwide), we probably have to provide some sort of automation. If the automation is to be effective, it must produce a large search space and provide a substantial degree of confusion, and diffusion. This then is the goal of evolutionary defenses.

Evolution can be provided in many ways and at many different places, ranging from a small finite number of defenses provided by different vendors, and extending toward a defensive system that evolves itself during each system call. With more evolution, we get less performance, but higher cost of attack. Thus, as in all protection functions, there is a price to pay for increased protection. Assuming we can find reasonably efficient mechanisms for effective evolution, we may be able to create a great deal of diversity at practically no cost to the end-user, while making the cost of large scale attack very high. As a very pleasant side effect, the ultimate attack may become necessary for each system under attack. In other words, except for endemic flaws, attackers may again be reduced to a case-by-case expert attack and defense scenario involving physical access.

1.3 What We Hope to Explore

In this paper, we look at how we can drive up complexity of attack with evolutionary defenses. We begin by looking at program evolution through several examples that show how complexity may be driven up. Next, we explore how evolution can be exploited to provide practical evolving defenses, and consider ways around evolutionary defenses. Finally, we summarize results, draw conclusions, and propose further work.

2 Techniques for Program Evolution

We will consider two programs equivalent if, given identical input sequences, they produce identical output sequences. The equivalence of two programs is undecidable as is the determination of whether one program can evolve from another [?]. This result would seem to indicate that evolution has the potential for increasing complexity of analysis, and thus difficulty of attack. In a practical operating system design, we may also have very stringent requirements on the space and time used by the protection mechanism, and certain instruction sequences may be highly undesirable because they impact some other aspect of system operation or are incompatible across some similar machines. For this reason, we may not be able to reach the levels of complexity required to eliminate concerted human attack, but we may succeed in increasing the complexity of automated attacks to a level where the time required for attack is sufficient to have noticeable performance impacts, even to a level where no attacker is able to design a strong enough attack to defeat more than a small number of evolutions.

We know that evolution is as general as Turing machine computation [?] [?], and that an exhaustive set of equivalent programs is easily described mathematically (i.e. the definition of equivalence), but this is not particularly helpful in terms of designing practical evolutionary schemes. We now describe a number of practical techniques we have explored for program evolution and some results regarding the space, time, and complexity issues introduced by these techniques.

2.1 Instruction Equivalence

In some computers, several machine instructions are equivalent. For example, there are several equivalent 'op-codes' [2] on the Intel 808x series of processors. An evolution can be attained by replacing applicable op-codes with equivalent op-codes. Assuming that 1 of every k instructions can be replaced by any of e different instruction codes in this way, we can produce e programs for every k instructions, or $(n/k)^e$ equivalent n-instruction programs.

Instruction equivalence, at least in the 808x family of computers has no impact on time or space, since equivalent forms of the same instructions operate identically. The effort required to make these transformations is minimal, and their impact on a serious attacker is relatively insignificant because detecting any of a set of op-codes is nearly as simple as detecting any particular element of the set. (i.e. at most a linear number of steps)

2.2 Equivalent Instruction Sequences

A very similar procedure involves replacing instruction sequences with equivalent sequences. For example, a sequence that adds 17 to a memory location can be replaced by instructions that add 20 and subtract 3, or any other combination of operations that yield the same result. This produces a potentially infinite number of equivalent programs. Another way to get equivalent sequences is by performing operations using different register modes, and saving and restoring registers respectively before and after executions.

The number of evolutions is potentially infinite in this technique, but it may also increase time and space. If we wish to use only the space equivalent replacements, we may be severely limited, while some evolutions may exchange time with space so as to make the resulting program either faster or slower.

In terms of attack, this process can greatly complicate things. For example, a test for '0' or '1' can be replaced by a large number of equivalent decision procedures. Several equivalent decision procedures follow:

[2] i.e. operation code - the portion of the instruction used to determine which machine operation is to be performed

	if $x = 0$ goto A		goto $x * 5 + . + 1$
	...	$. + 1$:	...
A:	...	$. + 6$:	...
	goto $x + . + 1$		$y = (x + 17) * 35$
$. + 1$:	goto A		if $x \leq (35 * 17)$ goto A
$. + 2$:	...		...
A:	...	A:	...

Clearly, we can derive enormous numbers of variations on this theme with little trouble. An attacker trying to determine what is being done in these code fragments has several problems. One problem is that any number of values may appear possible in x, even though, based on the original design, we may know that only the two values '0' and '1' are actually used. For other values, very strange behavior may result. Another problem is that as designers, we can control the entry points to these routines, whereas the attacker may not know all of them or understand what we knew as designers about the machine status at those entry points. By cascading these types of replacements, we can create enormous numbers of possible program executions, even though very small subsets are ever exercised due to external constraints on calling values. The complexity of attack may be dramatically increased while the difficulty of creating these evolutions is minimal.

Care must be taken in implementing evolutions of this sort at a low level because of the difficulty in identifying and maintaining instruction boundaries and entry points. In higher level languages, the problem is far less severe.

2.3 Instruction Reordering

Many instruction sequences can be reordered without altering program execution. In general, any linearly executed instruction subsequences that alter independent portions of the program state and output fall under this category. For example, assigning independent values to independent memory locations can normally be reordered. In parallel processing, it has been shown that programs produce a POset of dependencies, and programs for automatically analyzing these dependencies have been written to exploit this property in parallel processing applications. The number of reorderings is limited by the number of equivalent paths through the program, and the number of paths can be enormous. For example, a typical operating system call involves setting a series of values and making the call. In the setting of these parameters, order is rarely important, and we typically set two or three values, which means that each operating system call might be reordered into any of 6 different forms, not including any reordering of the calculations used for setting the parameters. In calls using arguments placed on the stack, we can similarly reorder the sequence of stack pushes and pops, again producing $n!$ orderings for n different arguments. Here is a simple example with 3 statements in all 6 orderings:

a=a+1; b=b*3+c; j=j+12;	a=a+1; j=j+12; b=b*3+c;	j=j+12; a=a+1; b=b*3+c;
b=b*3+c; j=j+12; a=a+1;	b=b*3+c; a=a+1; j=j+12;	j=j+12; b=b*3+c; a=a+1;

Reordering of instructions generally requires no additional time or space while providing $n!$ different variants, but this may not drive up the complexity of attack in cases where specific instructions are being sought for bypass. For example, the multiplication can be easily found in all six.

2.4 Variable Substitutions

In high level languages, we may use variable substitutions to alter program appearance, but this has little effect on lower level programs unless compilers produce resorted symbol tables. At lower levels, we may easily alter the locations of memory storage areas. By moving variables, we prevent static examination and analysis of parameters and alter memory locations throughout a program without affecting program execution. Here is an example of the impact of variable substitutions at the low level, where we have given numerical values to all instructions and labeled memory locations:

1:	2705	;jump to 5	1:	2705	;jump to 5
2:	0	;loc of 'A'	2:	1701	;loc of 'B'
3:	1701	;loc of 'B'	3:	2103	;loc of 'C'
4:	2103	;loc of 'C'	4:	0	;loc of 'A'
5:	1002	;A=A+1	5:	1004	;A=A+1
6:	2003	;B=B-1	6:	2002	;B=B-1
7:	2103	;shift B left	7:	2102	;shift B left
8:	1734	;xor C with B	8:	1723	;xor C with B
9:	4306	;B=0?goto 6	9:	4206	;B=0?goto 6
10:	0	;halt CPU	10:	0	;halt CPU

To see this effect more plainly, we now show only the values of memory locations and leave out the comments:

1:	2705	2705
2:	0	1701
3:	1701	2103
4:	2103	0
5:	1002	1004
6:	2003	2002
7:	2103	2102
8:	1734	1723
9:	4306	4206
10:	0	0

In general, we can place each variable at any program location not yet used by another conflicting variable. This yields $n!v$ different configurations for v variables in n program locations. In practice, variables are commonly kept in specific areas, stacks require allocatable areas, and other restraints on memory locations are common. By altering this practice and placing variables pseudo-randomly throughout the program, we may cause a great deal of diffusion. This is easily done by a compiler.

2.5 Adding and Removing Jumps

Many program sequences can be modified by placing a series of jump instructions where previous instruction sequences were located, relocating the previous instructions, and jumping back after sequences are completed. This produces arbitrary reordering of instructions, and at least $n!$ unique sequences for an n-instruction original sequence. Here is a simple example:

A:	I_1	A:	Jump A'
B:	I_2	B:	Jump B'
C:	I_3	C:	Jump C'
D:	I_4	D:	Jump D'
E:	I_5	E:	Jump E'
F:	...	F:	...
		B':	I_2;Jump C
		E':	I_5;Jump F
		C':	I_3;Jump D
		A':	I_1;Jump B
		D':	I_4;Jump E

We can similarly remove existing jump instructions and reorder programs, so long as no other part of the program transfers control into those sequences. Even in those cases, we can sometimes alter the jumps into the reordered sequences.

The addition of jump instructions increases space and time approximately linearly with the number of jumps inserted, while their removal increases performance and decreases space. This does not alter the form of specific instructions being sought, but it does a good job of obscuring program sequences.

One major problem with this technique is that different instructions take different amounts of space on some processors. This makes incoming jump instructions error prone unless the evolution process is done very carefully. At the source code level, the technique is quite simple, and thus it is particularly useful when we can reassemble or recompile code.

By designing programs with 'jump tables' or other similar structures, we may easily combine reordering of instructions and instruction sequences with altered jump tables to evolve programs in this way. A similar technique has been used by some software developers to make tracking of corporate licensees easier. When a particular evolved version is found, the corporation that released it can be easily identified.

2.6 Adding and Removing Calls

Programs that use subroutine calls and other similar processes can be modified to replace the call and return sequences with in-line code or altered forms of call and return sequences. Similarly, any sequence of in-line code can be replaced by subroutine calls and returns. Assuming that any single instruction can be placed in or removed from a subroutine and that there are k different subroutine call and return sequences, an n-instruction program results in n^k different evolutions. The time and space alterations are similar to those for inserting and removing jump instructions except that calls take more instructions to implement and may impact the stack and other processor information. Here is a simple example of call modification:

	goto Start;		goto Start;
f(a):	return(2*a);	o():	z=z+3;
pr(a):	wait-for-printer();		return(0);
	print(a);	Start:	x=3;
	return(0);		y=2*x;
Start:	x=3;		o();
	y=f(x);		x=z+y/2;
	z=z+3;		wait-for-printer();
	x=z+y/2;		print(x);
	pr(x);		halt;
	halt;		

Another important aspect of call insertion and removal is that it obscures high level design structure, and thus makes tracking similar operations more complex. Analysis of structure has been used by instructors in computer science courses to detect cheating, and this technique would likely invalidate those methods for use in automatic detection of program similarity.

As in the case of jump insertion and removal, it is far easier to perform call insertion and removal with knowledge of program structure.

2.7 Garbage Insertion

Any sequence of instructions that are independent of the in-line sequence can be inserted into the sequence without altering the effective program execution. Every instruction can have an arbitrary number of garbage instructions inserted, so there is no limit to the number of equivalent programs we can generate. [?]

Each added instruction increases both time and space, but is valuable for fooling programs that look for specific instruction sequences. For example, an attack that looks for IO calls to the operating system could be fooled by inserting spurious calls, thus forcing the successful attacker to examine more and more of the parameters used in the defense, and possibly increasing the time required for an attack to an intolerable level. As an example, we offer the following listing:

```
Start:  a=21; b=-19;
        for d=b to 98 step 7 do
        c=c+d;
        OScall(a,4,d);
        done
```

In this listing, we will now claim that the actual operation being performed is OScall(21,4,2), and that the other OScall invocations have invalid parameters, and thus return failures. The assignment of *a* is unrelated to this call, as is the repeated addition of d to c. In fact, the whole loop is unneeded, but this insertion of instructions is used to complicate the process of analysis.

By making spurious calls and using their results, we may greatly complicate analysis. For example, we could optionally call one of *n* different equivalent routines to make a decision, thus forcing multiple unique but equivalent paths through the program and making complete analysis far more complex. Again, the insertion creates confusion and diffusion. There is no limit to the amount of garbage we can insert, and a very broad range of relative ratios of garbage and program are available. The total number of different programs possible through garbage insertion is limited from above by the total number of free bits available for program space, which is limited only by available memory for program storage, and is clearly enormous.

2.8 Program Encodings

Any sequence of symbols in a program can be replaced by any other sequence of symbols, provided there is a method for undoing that replacement for the purpose of interpretation. For example, a trivial 'exclusive-or' (i.e. XOR) with a set of randomly selected bits stored in memory produces a random set of instructions, which can be recovered by performing another XOR with the same set of bits. Two common sorts of encoding schemes are compression and encryption. In compression, we find a coding that removes redundancy from the instruction sequence, and replace the original with its compressed form. In encryption, we find an encoding designed to obscure the content of a sequence. The decoding process reverses the encoding prior to execution. The number of encodings of a sequence are equivalent to the number of different sequences, or 2^n encodings for an n-bit sequence. A very good example of using encoding in an attack is given in [**?**], and this technique might also work well in defense.

The performance of coding schemes varies dramatically, and in the case where they must be decoded during operation, this can produce substantial time and space impacts. Furthermore, and attacker could wait till decoding is completed before bypassing protection unless the coding scheme varies as well as the things being encoded.

Encoding is strong against attacks involving examination of code, but if the decoded form is identical in all cases, it may be simple to find a way to alter the program after decoding or to observe the decoding key as it is entered or stored. Thus the use of coding alone is not sufficient for defending against serious attacks, even though it may help prevent the detection of a particular version by examination, as in the case of evading a virus scanner.

2.9 Simulation

Any sequence of instructions can be replaced by an equivalent sequence for a different processor, and that processor can be simulated by an interpretation mechanism. For example, we can invert all of the bits of a program, and simulate a bit-wise inverted version of the original processor. Any coding scheme can be used for this purpose, including a scheme that varies the code with the location. Again, there are at least 2^n different encodings for an n-bit instruction sequence.

In the case of simulation, we may have significant advantages in that the attacker must understand the simulation system as well as the machine language being used, or somehow detect the desired part of the code at run time. Unfortunately, simulation requires substantial time and space, and can only be used in circumstances where time and space are non-critical.

Here is a partial example of two evolutions of a simulator, where the ‘case’ statement is used by the simulator to determine how to interpret instructions, and instructions follow the ‘GO’ label as pairs of ‘op-code’, ‘argument’:

Loop:		Loop:	
Case	0:(op code 0)	Case	17: (op code 17)
	...		...
	6:(op code 6)	Case	8: (op code 8)
	...		...
	4:(op code 4)	Case	12: (op code 12)
	...		...
...			
Esac			
	goto Loop		goto Loop
GO:	0,12	GO:	17,12
	4,17		12,17
	6,17		8,17
	4,3		12,3
	0,4		17,4
	6,6		8,6

In our example, the ‘case’ statements of the two examples are aligned together so that you can tell the equivalence between op-codes, but in the second parts of the listings, you can see the impact of the transformation of op-codes on reading a program listing. In the following listing, we expand on this theme by a simple transformation of the arguments (op-code 17 subtracts 1 before equivalent execution, op-code 12 adds one, and op-code 8 subtracts 4). Now, there are no common byte sequences between the two program fragments.

GO:	0,12	GO:	17,13
	4,17		12,16
	6,17		8,21
	4,3		12,2
	0,4		17,3
	6,6		8,10

By implementing an evolving simulation engine for critical parts of the program, we can make detection of equivalence require both understanding the difference between the two evolutions of the simulation engine, and understanding the equivalence between the two sequences being simulated. This can be carried to multiple levels, with the simulator simulating another simulator, etc. until we get to the level at which actual decisions are made. We can even evolve the number of levels of simulation used.

It is very easy to implement a simulation engine in most current computers through the built-in debugging mechanism. The example above where we use a loop around a case statement shows just how easily simulation can be implemented with classical techniques, and in practice simulation engines require only a few thousands bytes of source code. Evolving a simulation engine and the simulated code is also very simple. For example, in the case above, it required only simple text replacement in the interpreted code and minor changes to the engine. Another simple alteration would be to XOR the memory value with a pseudo-random function of the location number in memory. This greatly increases confusion at the expense of only a small amount of time and space.

2.10 Build and Execute

An extension of the encoding and simulation techniques above is the 'build-and-execute' mechanism wherein we build instructions prior to execution and then execute them. This is so-called self-modifying code, and again the potential complexity is equivalent to the complexity of encodings. One of the very nice points about building instructions for execution is that it obscures the instruction being executed from observation until just before execution time. This drives the time required for attack up so as to make it very noticeable in many cases.

As a method of evolution, this technique requires that we design a set of build-and-execute mechanisms and install them into a program using some sort of pseudo-random selection and placement mechanism. As in the cases of many other mechanisms we have discussed, knowledge of the program being evolved is quite useful in implementing this sort of evolution.

Many build-and-execute schemes implement very complex codings that vary with use so that the instructions built may change with the details of the execution. For example, the use of relative addresses 'XOR'ed with instructions and filling in arguments at run time are common techniques.

Here is a simple example of self-modifying code, where the 'Add' instruction is modified to add each of the elements of a list of 17 numbers. In this example, we assume that the 'op-code' for 'Add' is 2300, and that the address being added is stored in the last two digits of the add instruction.

```
          AI=2299+@list                     ;initialize Add Instruction
loop:     AI=AI+1                           ;increment pointer
AI:       0                                 ;initial value is 0
          (AI—100) < list+16, goto loop     ;loop
          DONE                              ;whatever follows
list:     12                                ;the list to be added
          17
          ...
list+16   43                                ;end of the list
```

Assuming that the attacker has to observe these modifications at run time in order to determine when a particular operation takes place, an automated attack would apparently have to trace program execution and react to the particular instructions being interpreted under such an evolutionary scheme rather than searching for particular strings in a program or perform analysis from the program's appearance in memory.

2.11 Induced Redundancy

The use of redundancy is a basic technique in integrity protection. For example, the use of cryptographic checksums for detecting arbitrary corruption applies redundant information in the form of the stored checksum value, and CRC codes and parity bits are redundant information commonly used to detect corruption due to noise of particular characteristics. The same concept can be applied to preventing attacks on programs.

In this case, we induce redundancy by repeating test procedures used to make decisions so that multiple tests must be passed in order for an operation to be accepted. When used in conjunction with other evolution techniques, this provides a means by which the defender can prevent the attacker from being certain that the defense has been bypassed. For example, if a particular test is performed a random number of times on any given attempt at attack, the attacker cannot be certain that all of the relevant tests have been bypassed. Even if the attack works on one evolution, that doesn't mean it will work on a significant number of other evolutions.

In the following example, we show a redundant overall test which requires 3 out of 4 partial tests strewn throughout a program to be passed in order to pass the overall test.

```
count=0;
...
x=open-file('zz')
...
if (x<3) count++;
...
if (x>0) count++;
...
if (count < 1) OR (count > 2) nogood;
...
x=open-file('zz')
...
if (x = 2) OR (x = 1) count++;
...
if (count < 2) OR (count > 3) nogood;
...
x=open-file('zz')
...
if (x AND 3) > 0 count++;
...
if ((count AND 4) != 4) nogood;
...
```

A few notes are in order here. Notice first, that there is no explicit 'good' decision, but rather a set of hurdles that have to be passed and that are strewn throughout the program. If any of these tests fail, the 'nogood' decision is made, but there is no simple way to avoid the last 'nogood' decision. If there were an explicit 'good' decision, then an attacker could simply try to locate the 'good' decision point and begin operation there. Also, if the decisions were not strewn throughout the program, an attacker could easily bypass the whole set of decisions, but by intermixing them with the rest of the program, we force the attacker to work harder. Another important point is that there is no value for 'count' or 'x' that the attacker can select at the start of the program that will bypass all of the tests. This means that there is no simple way to forge the values required to make tests pass. Multiple forgeries are required. We could also use different variables for different parts of the tests, and further drive up the complexity of the attack process.

In practice, it is fairly simple to devise a series of tests and disperse them throughout a program, but ultimately tests depend on the state of the system for their accuracy, and if we are to eliminate all sorts simplistic forgery, we must not only make the tests redundant, but also the data upon which they depend. This increases the time and space requirements for accuracy, which yields our usual protection tradeoff between integrity and space/time.

2.12 Intermixing Programs

A rather complex sort of evolution is the intermixing of programs so that instructions from two independent operations are intermixed. This is complex to do because there are many possible interactions between memory and register states. In practice, we have found it far too complex to analyze and implement this sort of evolution at low levels, but in higher level programming languages, we see it as having great potential.

In this example, we have two subroutines that are often called, and which leave their results in independent variables 'x' and 'y'. As an evolution, we simply intermix the routines so that both functions are performed when either is desired, but no interference results because the results are stored independently, and by convention, we use or store the results of these routines immediately after the routines are called. Subroutine 1 is called with two integers as its argument, while subroutine 2 is called with an integer and a real number:

Subroutine 1	Subroutine 2	Mixed Subroutine
s1(i,j):=	s2(i,r):=	sb(i,j,r):=
x=0;		x=0;
x2=17;		x2=17;
	y=i+12;	y=i+12;
if (i<3) x=x+6;		if (i<3) x=x+6;
	y=y*r/3.74;	y=y*r/3.74;
x=x*i+j/17;		x=x*i+j/17;
return;	return;	return;

This sort of intermixing creates two problems for the attacker. The first problem is that the intermixing could be varied so as to produce a substantial number of different orderings of the statements in the subroutine. We can select the next statement from either subroutine, so long as there are unselected statements from both remaining, which yields on the order of 2^n different equivalent subroutines. The second problem is that the attacker cannot tell which function is being used when it is called, and may thus be forced to trace the implications of both calls through several following program steps before realizing what information can be ignored.

In general, we can take n unrelated programs and, assuming we are a bit careful, mix them together without ill effects. This seems to be very confusing to someone trying to analyze the resulting code, especially if the routines that are intermixed are selected at random for each defense. The mixing process also has the effect of disguising the subroutine calls because the calls themselves are different for different intermix combinations. In our example, the two mixed routines had two arguments each, and the mixed routine had three parameters.

2.13 Anti-Debugger Mutations

In order to make debugging programs difficult, many different techniques have been developed within the computing industry. The basic principle is to make the things the debugger does to track program execution fail when debugging the evolved program. Since each processor has a somewhat different debugging mechanism, we will make assumptions designed so as to make the examples easy to understand.

Disassemblers in systems with multiple instruction lengths have methods for synchronizing instructions. When encountering a jump instruction, typically, disassemblers assume that the following instruction is a legal instruction, but we can use these instructions to mask instructions around them by placing misleading operation codes after jumps, and returning at an offset 1 or more bytes from the end of the jump. The disassemblers may see a conditional jump (that is designed to always be true in actual operation) followed by an unconditional 3-byte jump which actually masks a 2-byte operating system call. Here is an example from an 8086 processor that calls 'int 13':

```
go:  jne 1         ;two byte conditional jump (flags clear)
     jle cd        ;conditional jump masking 'int'
     adc (13) 90   ;add carry 90 masking 13 and no-op (90)
```

An attempt at disassembly might fail in this context, but a debugger which observes each instructions as it is processed should be able to properly detect the operation.

In a similar fashion, we can include jump instructions whose last byte (usually the address jumped to) corresponds to a desired operation code, and place the code jumped to appropriately so that the jump works and the proper return address is in the middle of the previously executed instruction. In this way, we reuse the last bytes of the jump location as operation codes on the next pass through the code.

This technique again masks the true instructions being executed until execution time, thus forcing the debugger to trace each instruction in order to find a particular instruction type.

Debugging can often be turned off and turned back on in such a fashion that the debugger cannot tell that any change was made. From all appearances, the debugger executes normally, and yet some number of intervening instructions may go unobserved. This depends on the ability of the debugger to detect attempts to turn it off. We may use any addressing mode of the processor to alter the memory locations used by the debugger, so in order for the debugger to be certain to catch all attempts to access its address space, it must either simulate the entire operation of the computer, or determine all addresses used on each memory operation of the program at run time, determine whether those addresses impact the debugger, and forge the instruction if needed. Either of these options consumes a great deal of processing power and results in substantial time consumption, normally within the range where people notice the slowdown.

Another common trick is to make the operation of the defense depend on the presence of it's own debugging routine. For example, we could use the address of a built-in debugging routine or some offset from that address in an arithmetic operation which alters pointers, so that if the debugger operated by the defense is not operating (or alternatively, if any other debugger is operating), the code operates improperly. Another variation is to have the internal debugger alter normal instructions at execution so that (as an example) 'move' instructions that copy information between memory locations use different memory locations, reverse memory locations, alter operating modes, etc. This is essentially a simulation technique as described earlier.

Here is a simple example, again from an 8086 processor, of an instruction sequence containing illegal instructions and 'no-op' instructions, where the 'debugger' simulates altered instructions when these are encountered. Note that this is a common technique used by operating systems for operating system calls, and thus is also a call insertion technique as described earlier. In this example, the no-op instruction shifts the 'ax' register left 3 bits, and the illegal instruction (designated 'ill') sets the value 20 in the 'bx' register. Another debugger trying to debug this code would not alter the execution, and thus the program would not operate in the same manner.

```
go:  mov ax,3    ; initialize ax
     nop         ; no-op instruction (shift ax left 3)
     add ax,21   ; regular instruction
     ill         ; illegal instruction (setq bx to 20)
     add bx,3    ; regular instruction
```

All of these techniques can be selectively inserted and evolved during program evolution so as to produce a very complex debugging problem for the attacker. Even though some human attackers might eventually be able to get past this line of defense, they might have to observe a large number of different variations in order to determine the whole set of debugger bypass mechanisms, and then they would have to program attacks for all of them into an automated attack in order to operate automatically against this technique.

2.14 Mix and Match

All of the techniques described above may be mixed together, applied in any sequence, and applied recursively, thus providing a very rich environment for evolution. The only limitation is that identifying which code sequences may be operated on by each technique may be quite difficult unless the evolution engine knows something about program structure. For example, when a program is designed with numerous jump instructions that enter into the middle of other code sequences, it may be very difficult to determine whether an alteration will have an impact on some other execution sequence. This is the same problem we cause the attacker to go through, and as we have seen it can make things rather complex.

3 Providing Evolution in Defenses

As we have seen, considerable computation may be required in order to analyze a program in terms sufficient to perform evolution. This problem might be solved in several ways. The most obvious way is to mathematically formulate the conditions required for each sort of evolution being considered and evaluate a program to determine when and where these conditions are true. The problem with this sort of analysis is that it is, in general, undecidable whether most program properties are true.

To see this, we take the example of determining whether or not a program performs a jump into a particular location. This would be important to making an alteration such as random instruction insertion, since we cannot safely move instructions in such a manner that the incoming jumps have no altered effects. In general, we can only assure equivalence by determining where these entry points are and relocating the incoming jumps to fit the other alterations. The problem is that we can make determining whether or not a jump occurs equivalent to solving the halting problem as follows:

$$P := \text{If } J(P{,}x) \text{ then halt, else jump-to } x; \ldots$$

In this case, the decision routine '$J(P,x)$' determines whether program P jumps to location x. If $J(P,x)$ determines that P does jump to x, then P halts, and thus J is wrong. On the other hand, if J decides that P does NOT jump to x, then it does. Thus, J is always wrong, regardless of how it works, or in other words, you cannot write a J of this sort that works.

Naturally, this works for the attacker in the same way as the defender, and it is this sort of complexity that we are exploiting to make attack complex, but in order to perform the evolution for defense, the defender must be able to act with relative safety and in order to be practical, safe evolutions must be determined rapidly.

Very similar demonstrations can be made for many of the techniques we have discussed. Specifically, the equivalence of evolved instruction sequences is undecidable [?] [?] and the equivalence of machines [?] and therefore of simulations is undecidable. Variable substitutions, instruction reordering, adding and removing jumps and subroutines, and detecting garbage insertion appear to produce decidable, although in some cases nontrivial, equivalences.

The solution we have explored is to provide some additional information that can be used by the evolution system during evolution to identify when and where code can be altered in particular ways. These 'markings' are removed after evolution so that they are not available to the attacker in the modified form of the program, since they might serve as indicators of the relationship between the original and modified forms of the program. We call the fully marked version of the program a 'template'.

3.1 Avoiding Attacks

We must assume that the attacker has the same information as the developer of the defense because even the developer who writes a defense may eventually take part in an attack. We may keep the template in encrypted form on-line to further complicate the attack process, but it is certainly possible for an attacker to eventually gain access to the template. For these reasons, any method which is to have real strength must base its strength on the complexity introduced by some information only available to the evolver during the evolution process.

For these reasons, we will suppose that only pseudo-random numbers used to determine how to perform evolution are exclusively available to the evolver. Assuming we can reliably generate hard-to-guess pseudo-random numbers, [3] we then have the problem of assuring they are only available to the evolver. In order to assure this, we may take the strategy of building an evolver that operates at program installation, and generates a pseudo-random number based on various times and dates, keys provided by the user, and pre-existing system conditions. The evolver generates these numbers on the fly and does not remember them after they are used. Thus, in a normal installation, only the evolver ever has the information about which technique was applied in which way and under which circumstance, and that information is lost by the end of the evolution process, leaving only the effect of applying those numbers, and not the numbers themselves or other intermediate information used in the evolutionary process.

In order to bypass this constraint, the attacking program must be active before the defense is installed, but since this defense is designed as a part of the operating system, it would be very unusual for other programs to be operating prior to and during the installation process. This is particularly useful if included in the operationg system distribution, or if used just after a 'cold boot' from a known 'golden unit' version of the operating system.

As a side note, even knowing what sorts of evolutions are performed on which portions of a program and in which order the evolutions are done, does not make it trivial to map their effect in order to make meaningful modifications to the evolved version. For example,

[3]There are several well known techniques for this, the most common ones based on repeated exponentiation in a large modulus composed of large primes (i.e. based on the RSA cryptosystem [?]).

an evolution that generates a table look up based on parameter values which only apply to a small number of possible values, requires considerable effort to analyze. If this is done after another evolution, the effort becomes far more complex, because you may have to associate operations across numerous evolutions. The attacking program might have to remember a great deal of information or consume a great deal of time in order to determine hos to make such a modification.

3.2 Mixing Techniques

One key to building effective evolution in defenses is finding an appropriate mix of techniques and when to apply them so that performance requirements are met, while attack complexity is driven up. For example, we might include a call transformation at the factory, an encoding scheme carried out at product installation, jump table and sequence equivalence schemes at system start up, and instruction building during each system call. This mix may minimize performance impact while providing high complexity for attacks across systems. Other mixes may be more appropriate for different environments.

Building an evolutionary system into a defense, however, is not quite as simple as implementing a set of evolutionary schemes and putting them into a product. Reliability and repeatability are important factors, and great care must be taken to assure that the end user isn't impacted by the evolutions or the evolutionary process. Debugging can be quite difficult, since the operation may change as the program operates, and you can never be absolutely certain what you are debugging unless you create appropriate tools for the process. The tools you devise for debugging may also be used by an attacker, so they too must be considered in analyzing the strength of the defense.

3.3 Selecting the Mix

As a design strategy, we normally trade time and space for protection, so it seems reasonable to devise a method wherein we specify final time and space usage, provide a template, and tell the evolver to evolve until the time and space usage are met. We may then categorize evolution techniques in terms of time and space tradeoffs, and subject to statistical variation, use techniques in proportion to their impacts on these factors. This allows us to generate a mix of techniques that meet the desired tradeoffs, but it doesn't force substantial evolution. To assure substantial evolution, we might specify some minimum evolution requirements, and fail due to lack of time or space if these minimums cannot be met.

The degree to which we attain protection is clearly dependent on the amount of free time and space provided for evolution. By our estimates based on speculation and nominal experience, a 50% space increase has relatively little impact on performance while providing a great deal of obscurity, but clearly more work is required in this area before we can draw conclusions.

A simple algorithm for controlling evolution based on the idea of limiting the size of the final product and using different types of evolution with varying probabilities follows. In this case, we have assigned the evolutions (evolution$_1$ ... evolution$_n$) probabilities (prob$_1$ ... prob$_n$), and perform evolution based on pseudo-random numbers:

```
outer:  if total-space > threshold-space goto end;
        x=0;y=0;
        a=pseudo-random number between 0 and 1;
inner:  y=y+1;
        x=x+prob_y;
        if x < a goto inner;
        do evolution_y;
        goto outer;
end:    exit;
```

3.4 When to Evolve

Another important problem to consider is when to evolve. Although we don't have a closed form solution to this problem, we believe we have found some rational principles. Again, we want to encourage other researchers to look at these problems in more depth.

Evolution 'at the factory' before sending out each disk is a rational approach which has several advantages and disadvantages. The major advantage is the ability to uniquely identify each disk sent out from the factory. This allows tracking of unauthorized distributions to their source, provides a means for generating unique registration numbers (e.g. a cryptographic checksum of the programs) which can be verified by the programs themselves at various points and by the manufacturer to assure that no corruptions have taken place in distribution. This also increases the complexity of attacking the original distribution, since it is unique for each disk.

On the negative side, production efficiency is dramatically reduced when we cannot simply duplicate each disk, especially when strong quality control procedures are used. For example, we may test every disk prior to sending it to manufacturing and perform statistical tests on disks after duplication. When using evolution at the factory, we dramatically increase the costs associated with equivalent quality control. A tradeoff at the factory is common, where some people use evolution once for each production run, or produce evolutions for each large customer.

Evolution at installation is critical if each system is to have a unique and confidential evolution in place. An evolution at the factory or in the distribution path could be copied or found by an attacker. After installation, an attacker could prevent further evolutions or store the internal values for the evolution process. For these reasons, some evolution at installation is important, but evolution also takes time, and in the current computing environment, time consumed at installation is treated as a serious negative by most customers. One way to perform evolution without incurring much customer wrath is to put up screens full of information at various points during the installation process, and require user input at those points. While the user figures out what to do, evolution can be performed. The same technique can be used while waiting for the insertion of a floppy disk or during other human operations.

Evolution after installation produces several problems, most notably performance problems and integrity checking problems. When we produce evolutions at the factory, we can create integrity information and provide it to the customer. When the customer installs the product, they can verify the integrity prior to installation, do installation along with its associated evolution, and produce new integrity checking information for subsequent use. When we evolve on every system start up, we cannot use external checking to assure against corruption, and we cannot trust internal checking because it could have been corrupted. The major advantage of evolution after installation is that an attack which succeeds on any one day might fail on the next day if the attacker has not been able to sufficiently corrupt the mechanism so as to produce a bypass under multiple evolutions or so as to prevent further evolution completely..

It seems clear that the more time and space we have available for evolution, the more we can obscure the evolved program. It is also clear that different amounts of time and space are available at the factory, during installation, during start up, and during operation. For a small resident program operating under DOS, we can hardly put in a great deal of self-evolution capability; while at installation, we may be able to evolve the installed version without even copying the evolver onto the system; and at the factory, we may be able to run very fast computers with a great deal of storage for a very long time in order to produce a single evolution for one large corporate customer.

3.5 Experiments With Evolutionary Defenses

We performed a number of experiments with evolutionary defenses. The first step was to write a general purpose program evolver which we could apply to test the viability of the techniques under consideration and understand their interactions and how well they obscured the meaning of a program to an experienced programmer. In this sense, the general purpose evolver was a tremendous success, but program evolution also presented several problems.

The first major problem in program evolution is that in order to do the more appealing sorts of program evolution, a great deal of program specific knowledge is required. If we were to analyze a program from the assembler level and try to determine what it did in order to determine where to make evolutions, we would be faced with a horrendous problem. We would have to be able to reverse engineer the executable program to the point where we could show each of the properties required for each form of evolution over all program subsequences, and then apply the techniques to the applicable subsequences. In order to do this recursively, we would have a major mathematical problem in determining which techniques could then be applied to the partially evolved program. Even for classes of programs obeying very stringent rules this seems to be a major problem. This is in keeping with our previous theoretical results, and therefore this is not too surprising.

We decided to provide annotations in source programs that indicate the extent over which each technique can be applied. We created a Lisp program that reads the annotations and calls routines that implement each type of evolution. We then compiled the evolved sources to produce compiled evolved executables. Thus, we are designing for evolution. As an example, a code fragment and one evolution are show below:

```
:beginGarbage                              | real            oou, oi, ou, oj;
              for(i=0;i¡99;i++) {          | subroutine ox()
:beginCalls                                |                 {c=d+e;
              a=b+5;                       |                 oi=23;
              c=d+e;                       |                 b=a+i*c;}
              b=a+i*c;                     | subroutine ob()
:endCalls                                  |                 {x=2+y/z;
              }                            |                 ou=oi+12;}
:beginReorder                              |                 oj=23+ih;
              x=x&2;                       |                 for(i=0;i < 99;i++) {
              y=y-5;                       |                 a=b+5;
              z=xray(23);                  |                 ox();
:endReorder                                |                 }
:beginCalls                                |                 y=y-5;
              if (z<27) print(”Error 23”); |                 z=xray(23);
              x=2+y/z;                     |                 x=x&2;
              y=sqrt(23*a+c);              |                 if (z<27) print(”Error 23”);
:endCalls                                  |                 ob();
:endGarbage                                |                 oou=23*oi;
                                           |                 y=sqrt(23*a+c);
```

It is plain that the evolved version is quite different from the original, and this example only applies a few of the techniques above in a relatively simple manner, and over a very small fragment.

In our experiments with high level languages, we have found negligible performance impact except in inner loops, and with minimal effort, we can maintain performance by limiting evolution within these areas. In terms of space consumption, the impact is even less significant, but this can be controlled by the degree to which we introduce space consuming techniques. For example, the introduction of garbage statements into a program is performed with some finite probability between each two statements. Higher probabilities induce more space and time overhead, while lower probabilities result in less confusion.

The evolution process itself was quite fast, even though we were operating in an interpreted language on programs that had to be compiled after evolution. By way of a benchmark, we had no difficulty evolving a thousand line program and compiling the result in well under 1 minute. The implications for attack are quite stunning. It means that we could easily produce about 1,500 different evolutions per day of a program far more complex than a typical virus, each of which would require substantial effort to defend against by scanning for known versions. Even though this attack is easily defended against by the best existing defenses, the vast majority of the worlds computers using virus defenses are using techniques against which this would be quite effective. By way of perspective, in 1992, the total number of known virus variations existing in the world (outside of our laboratory) passed 1,500, while we have had the ability to generate more than that many new variations per day for several years.

We have tested and installed a partial variation on this theme in computer defense products for over 2 years. A typical memory resident protection system in a PC uses under 4Kbytes of memory (far less than the size of a typical compiled 1,000 line C program), and it is not terribly difficult to perform some simplistic evolutions at product installation time and on every system reboot. Among the techniques already in widespread commercial use are altering filenames of protection critical files for each installation, encrypting the contents of these files with system dependent keys determined at installation time, altering memory resident software at the factory and at installation time, and using anti-debugging techniques to prevent attacks that look for memory locations by simulating operation. We have shown feasibility for encrypting significant portions of the DOS operating system and decrypting it at execution time for use, and hope to have prototypes in testing fairly shortly. It appears that there are no significant impediments to the widespread application of these techniques other than our desire to apply them and pay the price in time and space.

4 Attacks on Evolutionary Defenses

As we discussed earlier, human attacks on evolutionary defenses may be possible with concerted expert effort and physical access, but the question that must be addressed in more depth is the difficulty of implementing an automated attack against a defense applying these techniques. Although we cannot yet address the specific complexity of attacks against these techniques, some results are fairly apparent, and we present them here.

4.1 Points of Attack

The most obvious point of attack is a particular instruction or operation performed by the program. For example, system calls are generally standardized, and if the defense calls the original system call after checking parameters, it might be easy to find the original entry point and exploit it for attack. This is done by several computer viruses, which attempt to bypass controls by direct access to the hardware or operating system internals. One defense is to evolve the core of the operating system, while another is to confuse attack by evolving the calling mechanisms. In the latter case, calls that bypass protection may be of the wrong form unless significant attack effort is applied to determine and apply the altered calling sequence.

Since we may sustain low-level attacks, normal operating system protection does not help in this matter because an attack introduced before the operating system is started can bypass even sound operating system controls on processors with hardware memory protection. Low-level attack with direct access to hardware is straight forward, especially when system structures are known to attackers. For this reason, any system defending against this sort of attack must apply some technique to prevent an attacker from examining and modifying the protected information or simulating the entire operating system without protection. Even with low-level encryption, the encryption software itself must be evolved in order to prevent an attacker from finding the key used for encryption or forging the I/O used to get the key from the user.

4.2 Tracing Attacks

As we described earlier, tracing programs at execution time or simulation program execution can ultimately result in precise information about a single program execution. A perfect simulation cannot be avoided, and a human attacker would likely find a way around any evolution, given enough time and persistence. The use of redundancy appears to be the only effective way to force the attacker to use tracing on each attack and repeatedly during normal operation in order to systematically bypass defenses.

4.3 Size of the Space versus Complexity

A critical factor to understanding the real impact of evolution as a defense is understanding that generating a large search space does not necessarily result in a difficult attack problem. Just as passwords may be easily guessed even though the total number of possible passwords are large, a large number of evolutions does not necessarily make attack detection of a particular portion of a program difficult. For example, reordering a sequential series of instructions which are never entered from other portions of a program does not make detection very difficult, even though it may create any of $n!$ different program sequences. We can detect the sequence by searching for ANY of the instructions, and checking off that instruction from a list. This takes only linear time with the number of instructions in the sequence. We thus conclude that increasing the sheer number of possibilities does not necessarily increase the complexity of attack.

Even more importantly, we must be clear about what we are trying to prevent through the evolutionary process. We have been discussing the difficulty of determining the equivalence of two program sequences, but this does not necessarily correspond to the difficulty of attacking a system. For example, denial can be caused regardless of what the operating system does if we can sufficiently corrupt the contents of a disk.

4.4 Toward Computational Advantage

The ultimate goal of evolutionary defense seems then to be finding a way to modify a program so that the computation required for some set of attacks is sufficiently large as to make automation of these attacks infeasible. There is a tradeoff involving the amount of time and space required to perform evolution, and thus, in order for such a system to be rational, we generally must exploit some computational advantage. That is, we must find a way to perform a transformation whose meaning is hard to invert relative to the aspects important to the attacker, by making the effort required to bypass the transformation large relative to the effort required to perform the transformation.

The concept of creating computational advantage is not new. For example, the most commonly used cryptosystems can easily be broken given enough computation. The way they succeed is to keep the computation required to encrypt and decrypt with the secret key relatively low, while making the computation required without they secret key very high. In the case of program evolution, the goal is the same, but the circumstances require that the computer constantly be able to derive meaningful information in the form of proper program operation.

5 Summary, Conclusions, and Further Work

We have introduced the concept of using evolution to increase the complexity of attack, and we believe that this concept is sound and will have a lasting impact on protection in operating systems as well as in other areas of information protection, but clearly, our work has only begun with this effort. We have identified several methods of evolution that may be effective, and in some cases, have even shown that some properties of these techniques are undecidable, but we have found no firm basis for believing that these particular techniques will indeed be effective against serious human attackers except the very flimsy arguments about the complexity of program equivalence and conclusions about the size of the resulting probability spaces. As we have also shown, these factors may not indicate the difficulty of some sorts of attack.

The idea may be sound, and these techniques may be effective, but more mathematical analysis is required in order to assert a high degree of assurance in these notions. Indeed, a major stumbling block along the road to a mathematical analysis of the enhancement of protection through this technique is a mathematical understanding of protection issues and level attacks against mechanisms. As in the area of cryptography, there is rarely a sound basis for the practical techniques we apply, but we still have the notions of diffusion, confusion, and computational advantage to consider as possible metrics for evaluating efficacy. Perhaps the best positive conclusion we may draw from this work is that evolution seems to be practical, effective, and operable, and that it dramatically increases the workload of attackers relative to non-evolutionary systems.

Clearly, a great deal of further work is required in order for this field to mature. Specifically, more mathematical analysis of attacks and defenses, a better understanding of what we are trying to conceal and the degree to which evolution is effective at concealing it, and results on the tradeoffs of time and space of techniques are clearly called for.

References

[1] C. Shannon, "Communications Theory of Secrecy Systems", Bell Systems Technical Journal, 1949 pp656-715

[2] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems", CACM V21#2, Feb. 1978.

[3] "The Data Encryption Standard", National Bureau of Standards, Washington, D.C, 1980

[4] F. Cohen, "Computer Viruses - Theory and Experiments", DOD/NBS 7th Conference on Computer Security, originally appearing in IFIP-sec 84, also appearing in IFIP-TC11 "Computers and Security", V6(1987), pp22-35 and other publications in several languages.

[5] F. Cohen, "Algorithmic Authentication of Identification", Information Age, V7#1 (Jan. 1985), pp 35-41

[6] F. Cohen, "Computer Viruses", Dissertation at the University of Southern California, 1986. - originally published by ASP Press, 1985

[7] F. Cohen, "A Short Course on Computer Viruses", 1990 ISBN#1-878109-01-4, ASP Press, PO Box 81270, Pittsburgh, PA 15217, USA

[8] F. Cohen, "A Note On High Integrity PC Bootstrapping", IFIP-SEC "Computers and Security" (submitted, 1991)

[9] A. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", London Math Soc Ser 2, 1936.

[10] E. Wilding (Ed.), "Computer Virus Bulletin", many issues in 1990-92